# Robotics and Computer Vision 2 for UAS

UNIVERSITY OF SOUTHERN DENMARK

Miniproject four

Frederik Mazur Andersen
fande14@student.sdu.dk

Kristian Møller Hansen
khans13@student.sdu.dk

Olliver Ordell
olord13@student.sdu.dk

*Hand-in date: May 17, 2018*

# Contents

# 1 Introduction

This project is about locating a drone with video from a webcam, and controlling it's attitude through ROS. The objective is to position the drone at a fixed point relative to the stationary camera. This drone will be equipped with an ArUco marker, which has a relative easy interface with OpenCV, in order to locate it. The relative position in $x$, $y$ and $z$, the rotation in *roll*, *pitch* and *yaw* together with the ID of the marker is then published to a ROS-topic, which a controller subscribes to. The controller calculates the error, and sends corresponding commands to the drone via a FTDI-cable that transmits the signal the serial connection on a Arduino Nano. The Arduino then transmit the signals to the drone though a DSMX module.

# 2 Program structure

To get a better view of the system structure, a figure of the program flow is shown in figure 1. From the top left the video feed gets analyzed by the MarkerAttitude, which publishes the drones marker position, orientation and ID, on the ROS-topic `/markerlocator/attitude`. The CameraController subscribes to this topic, calculates the error and publishes the motor commands in % on the ROS-topic `/remote_control/set_controller`. The RemoteControl then converts this into PPM-signals that is sent to a Arduino Nano via a FTDI-cable. The Arduino Nano is mounted on a radio transmitter which then transmits the PPM-signals to the drone.

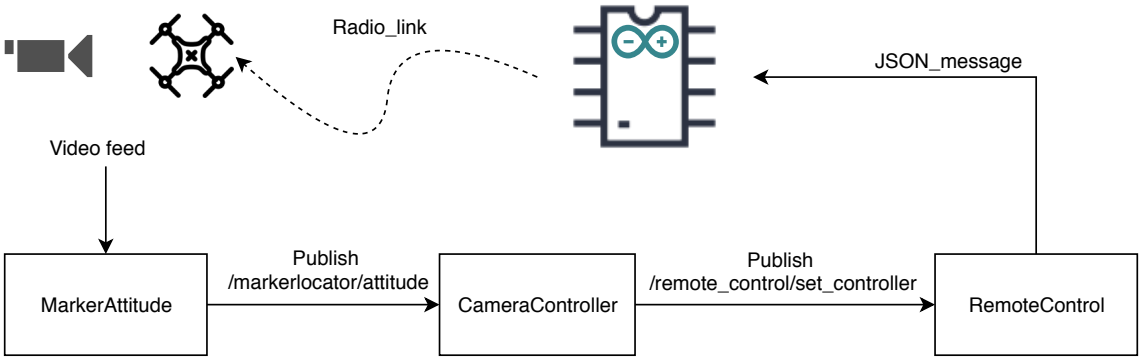


**Figure 1:** Illustration of the system structure and program flow.

# 3 Finding the drones attitude

In order to find the drones attitude, an ArUco markers was placed on the top of the drone. These markers can be generated easily via the library in OpenCV. By trying different physical- and dictionary sizes, the group found that a marker with $\sim 7\,\text{cm}$ in width and height, and a dictionary size of 7x7 worked best for this application. An example of this marker can be seen in figure 2.

## 3.1 Calibration of camera

In order to locate the markers correctly, it can be crucial to have a good calibration done for the camera in use. This was done with a chessboard and a python script found at

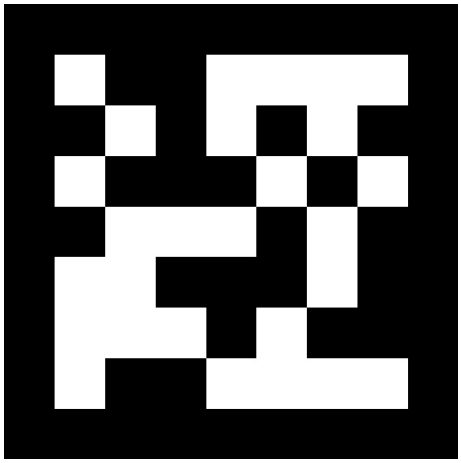

**Figure 2:** Example of a 7x7 ArUco marker.

`https://github.com/LongerVision/Examples_OpenCV/blob/master/01_internal_camera_calibration/chessboard.py`.

The group tried to raise the resolution of the video from the webcam, from the default of 640x480, however this resulted in a huge delay, which was thought to be more than the controller could handle.

Due to the webcam given it was necessary to modify its exposure. By default it was set to 667 which overexposed the image in the testing location and gave massive motion blur. This meant that if the drone and its marker moved faster than very slow, then the marker couldn't be detected. Changing the exposure setting to 170 made the video slightly underexposed and removed almost all the motion blur. This made the finding of the marker more reliable and made it possible to detect the marker at higher speeds.

### 3.2 Locate the marker in a video

To locate the ArUco marker, the ArUco functions from OpenCV is then used to locate and visualize the markers position. The function `aruco.detectMarkers()` is used to find the markers, and the functions `aruco.estimatePoseSingleMarkers()` and `aruco.drawAxis()` is used to draw the a box around the markers, and draw the markers coordinate system respectively. The program is set up so it can find multiple markers at the same time, each with there own unique ID, however only one marker was placed in the drone for this application.

The z-axis tend to flip in some cases, especially when the camera looks directly at the marker. This will wrongly change the error by 180 °. These flips is filtered out by checking if the change is greater than 10 degrees and checking that the found yaw angle are within the 180 ° the marker is visible in. The filtering is visible as short bursts of frames where no pose is found and displayed.

The program works surprisingly well in real-time, and can easily find multiple markers simultaneously with less delay then humanly observable. A video of this can be seen **here**. The video may seem to lag a little bit, but this only happened when the screen-recorder was on. The footage is from before the flips on the z-axis was filtered out. Note how the z-axis tended to flip, but only from one side. The z-axis also seems a little unstable when the marker is perpendicular to the camera.

# 4 Controller

The controller node uses information about the marker location and orientation to regulate the drone by trying to position it on a setpoint in the center of the image. This setpoint is defined as a 3D coordinate with an orientation around the drone's yaw axis. A service was also implemented that made it possible to change the setpoint while flying. The service is called on topic `/camera_controller/new_setpoint`.

In order to generate an appropriate command to the drone, the error signal from the marker has to be transformed to the perspective of the drone. The transform from the camera to the marker (eq. (1)) can be constructed from the pose information given by the markerfinder node, which gives you its rotation matrix. The transform from the marker to the drone perspective (eq. (2)) can be approximated by looking at how the axes of the marker lines up with the control axes of the drone. The final transform (eq. (3)) can then be calculated as the product of these two matrices. This transform is then used to convert the error signal into the perspective of the drone (eq. (4)). The orientation error is not transformed since it is a relative measure that stays the same in both perspectives.

$$T(R)^{marker}_{camera} = \begin{bmatrix} R[0,0] & R[0,1] & R[0,2] & 0 \\ R[1,0] & R[1,1] & R[1,2] & 0 \\ R[2,0] & R[2,1] & R[2,2] & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

$$T^{drone}_{marker} = \begin{bmatrix} 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

$$T^{drone}_{camera} = T^{marker}_{camera} \cdot T^{drone}_{marker} \tag{3}$$

$$error_{drone} = \left(T^{drone}_{camera}\right)^{-1} error_{camera} \tag{4}$$

The commands to the drone are based on the assumption that errors in the x and y-directions of the drone corresponds to roll and pitch commands, respectively, and errors in the z-direction correspond to a change in thrust. This is a reasonable approximation because there already is a low-level controller on the drone, that tries to maintain an approximately level orientation. The used flighmode for this project was chosen as `Horizon` which is self-stabilizing. The control loop consists of a possible PID controller for each of the four control axes; roll, pitch, yaw and thrust. However for pitch, roll and yaw only a P controller was needed and a PI controller is used for the thrust. The trust needs a PI as the possible thrust changes as the battery discharges and the controller doesn't use sliding mode. Each of these axes have to be tuned separately, because their dynamics vary quite a bit. The thrust, especially, is different from the others since it is also fighting gravity. The PI constants for each axis used for the resulting video in 6 is shown in table 1. The initial tuning was attempted by using the method of Ziegler-Nichols Closed-Loop Tuning.[1] This gave initial good results, but in order for the drone to not move too fast for the camera to track it, it was necessary to lower the PI values from the method quite a bit. The markerfinder also sends out information about the id of the marker which could be used for more precise localization of the drone as it turns away from the camera.

| Constants \ Axis | Roll | Pitch | Yaw | Thrust |
|---|---|---|---|---|
| P | 0.10 | 0.10 | 0.15 | 0.10 |
| I | 0 | 0 | 0 | 0.07 |
| D | 0 | 0 | 0 | 0 |

**Table 1:** Constants for the PID controller of each axis.

# 5 Linking controller with remote

The link from a controller to the transmitter and thus the drone consists of a interface between a rosnode called RemoteControl and its serial connection to the Arduino on the transmitter. This section will describe both parts and how they work together aswell as how to use it. The RemoteControl and Arduino controller is taken from another assignment in the course RMUAST where we created it and then improved for this application. This is also one of the reasons the interface is built up in this way, so it is easy to integrate and use in different projects and only change what interacts with the RemoteControl.

## 5.1 RemoteControl

The rosnode subscribes to a topic called `"/remote_control/set_controller"` with the message type seen in code 1. The floats are percentages. The `queue_size` used for the publishers is set to `1` to ensure that whenever newer data is available it will always get used instantly. This is to remove delay and minimize the latency for the controller.

The rosnode then checks the message and clamp signals that is outside the range, so it isn't possible to send values like 200 %. The message then gets converted to ppm based on the input percentage. Throttle goes from 0 % to 100 %, while the other channels goes from -100 % to 100 %, where the midpoint is 0 %. The midpoint matches 1500 ppm. The message with the ppm values gets packed into bytes which gets sent to the transmitter over serial connection. The message consists of four `shorts` with a combined size of 8 bytes. This make the message very small and super fast to transmit. To pack the message into bytes the python fuction `struct.pack()` function is used.[2] When this node is running, the Arduino only functions as a relay to the transmitter, which means that more advanced features like changing the sensitivity or other behaviors of the controller, can be integrated into the node to make it more easily configurable from the outside.

```
# This contains the percentage of stick movements for the channels
float64 thrust
float64 roll
float64 pitch
float64 yaw
```

**Code 1:** The message format used to send data to the rosnode that bridges the data into the transmitter.

## 5.2 RemoteControl to transmitter interface

The interface with the transmitter is done with a rosnode that communicate with the transmitter over serial connection, though a FTDI cable. The chosen format is four `shorts`

packed into bytes. When the Arduino recieves the communication it is unpacked into the four `shorts` and checked if they are valid ppm signals. This is done by the code seen in code 2. The values is given to the transmitter in ppm.

The Arduino transmitter implementation was changed to utilize a statemachine for better readability and performance. The statemachine can be seen in the file, or in the github repo [1]. This also solved an issue with emptying the serial buffer when changing from manual mode to offboard mode. As the Arduino have a serial buffer of 64 bytes it buffer 8 messages. This means that if the buffer is full and the mode is changed to offboard mode, then the new data won't appear for 8 messages. This created a delay that took time to catch up to, while giving too much latency to the controller. The solution was to have a intermediate state when moving from manual to offboard that cleared the serial buffer.

```
{
    const byte numBytes = 8;
    byte Buffer[numBytes];
    int bytecount = Serial.readBytesUntil('\0', Buffer, sizeof(Buffer));

    //highbyte then lowbyte
    unsigned short roll = Buffer[0] * 256 + Buffer[1];
    unsigned short pitch = Buffer[2] * 256 + Buffer[3];
    unsigned short yaw = Buffer[4] * 256 + Buffer[5];
    unsigned short thrust = Buffer[6] * 256 + Buffer[7];

    //check for limits on ppm
    if (roll > 1150 && roll < 1850 && pitch > 1150 && pitch < 1850 && yaw >
         1150 && yaw < 1850 && thrust > 1150 && thrust < 1850)
    {
        ppm[0] = thrust;
        ppm[1] = roll;
        ppm[2] = pitch;
        ppm[3] = yaw;
    }
}
```

**Code 2:** The Json message format used to send data over serial connection.

## 6 Results

Due to time constraints the drone didn't get fully tuned and it still didn't stay consistently in position in front of the camera. However at the end it really seemed like it was just a matter of tuning the controller better and not other bugs that cause the drone to fail. A video of the result can be seen **here**. As the video shows it do keep the position alright, but for thrust it overshoots and regulates too fast. This means that it moves out of the frame where it can't be tracked and that it then tries to compensate when it get back into the frame too rapidly. When the drones move too fast the camera can't track it. It seems to be an issue with the camera. If the exposure gets lowered more it can track it faster, but it gets too dark to see it when it moves a bit away.

The flight in the video is also shown in the graphs in figure 3. It shows that the thrust starts out alright, but gets into overshoots or overcompensation further in, where it then

[1] https://github.com/Crowdedlight/SDU-UAS-TX/blob/remotecontrol/python_controller/python_controller.ino#L188-L229

snowballs out of control. The pitch, roll and yaw was attempted to tune as well and only a few oscillations can be seen, but as thrust is still unstable it makes it hard to correctly tune for the other directions.

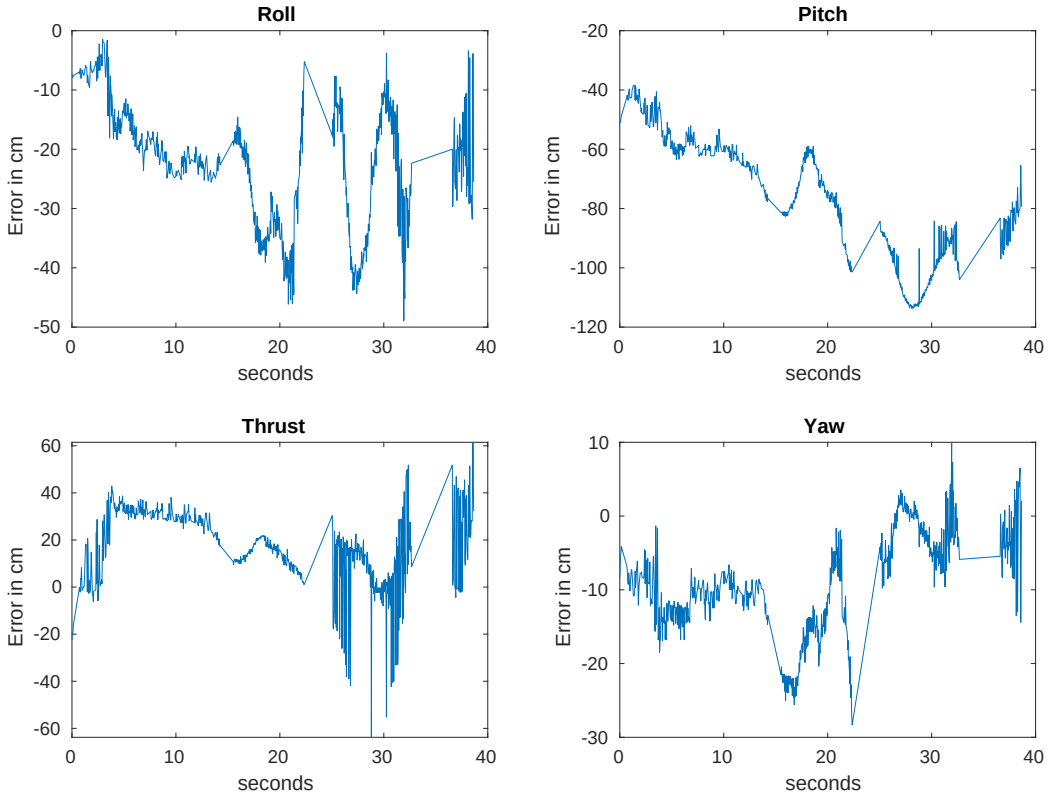


**Figure 3:** Error from each axis over time. Shows the error in cm, that the drone is from the target setpoint.

# 7 Discussion

The primary thing that didn't succeed was the tuning of the controller. After having issues with the delay and latency which resulted in erratic behaviour the entire Arduino code for the transmitter was re-factored and the format for serial communication was changed to use bytes for better performance. This greatly improved the behaviour and removed most of the inconsistent flying.

The camera turned up to be the bottleneck when tuning the drone. The camera limited how fast the drone could regulate itself as it didn't allow for the drone to move very fast. This meant that if the drone got into a situation where it moved too fast it would snowball and not be able to regulate itself. With a more delicate tuning and in general changing the controller to be more slow at regulating would fix this problem, however this also makes the drone very slow to react and thus easier to drift outside the seen field of view from the camera.

## 8 Conclusion

It was possible to make a system where the drone got regulated based on a marker and a camera locating the marker. The system worked in terms of communication and interface from computer to the drone. However the tuning of the P and PI controller didn't manage to get very good. This was an combined issue of difficulty in tuning and a camera that limited the possible movement speed. If given more time and possible another camera it should be possible to get the drone tuned correctly and actually stay in position.

## 9 References


[1] opticontrols. *Ziegler-Nichols Closed-Loop Tuning Method*. 2018. URL: `https://docs.python.org/2/library/struct.html`.

[2] Python. *struct — Interpret strings as packed binary data*. 2018. URL: `https://docs.python.org/2/library/struct.html`.